451-WDISC-PSSDG99

NETWORKS AND MISSION SERVICES PROJECT

WDISC
PTP Scheduling Server
Design Guide

Version 1.3

30 March 1999

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland

"The time has come," the Walrus said,
"To talk of many things:
Of shoes -- and ships -- and sealing-wax –
Of cabbages -- and kings --
And why the sea is boiling hot --
And whether pigs have wings."

Lewis Carroll

Contents

Section 1. Introduction

1.1 Purpose 1-1

1.2 Scope 1-1

1.3 Document Organization 1-1

1.4 References 1-1

Section 2. PTP Scheduling Server Design

2.1 Design Philosophy 2-1

2.2 Assumptions 2-1

2.3 Language 2-1

2.4 Coding Conventions 2-1

2.4.1 Files 1-1

2.4.2 Names 1-1

2.4.3 Comments 1-1

2.5 Compiler 2-1

2.6 Design Overview 2-1

2.7 Design Specifics 2-1

2.8 Implementation Details 2-5

2.8.1 Definitions 2-6

2.8.2 Main Routine 2-6

2.8.3 Database Class 2-8

2.8.4 Event Class 2-9

2.8.5 Header Class 2-9

2.8.6 Logfile Class 2-9

2.8.7 PTPServer Class 2-10

2.8.8 SchedEvent Class 2-10

2.8.9 SchedSocket Class....2-11
2.8.10 SortedList Class....2-12
2.8.11 Time Class....2-12
2.8.12 Timer Class....2-12
2.9 Known Bugs....2-15
2.10 Missing Features....2-16

Appendix A. Abbreviations and Acronyms

Appendix B. Windows NT Service Setup

Appendix C. PTP Scheduling Client Operation

Figures

Figure 2-1 PSS Class Interactions 2-5
Figure 2-2 Main Routine Flow Diagram 2-8
Figure 2-3 Scheduling Socket Flow Diagram 2-11
Figure 2-4 Timer Flow Diagram 2-13
Figure 2-5 Detailed Event Timing 2-15
Figure 2-6 Scheduling (Delay Calculation) Flow Diagram 2-16

Tables

Table 2-1 Naming Conventions 2-2
Table 2-2 Definitions 2-7

Section 1. Introduction

The White Sands Complex (WSC) TCP/IP Data Interface Service Capability (WDISC) supports customers who require TCP/IP access to the WSC for telemetry and command processing. Support is provided from the NASA Integrated Services Network (NISN) Closed IP Operational Network (IONET), using a defined set of authorized addresses. Support provided by the initial version of the WDISC is limited to three customers: New Millennium Program Earth Orbiter-1 (NMP/EO-1), Gravity Probe B (GP-B), and Far Ultraviolet Spectroscopy Explorer (FUSE).

1.1 Purpose

This document should serve to adequately describe the design of the WDISC PTP Scheduling Server (PSS) software to anyone who might need to maintain or enhance the source code. Furthermore, any questions that may arise about the day-to-day operation of the PSS are probably answered here.

1.2 Scope

An understanding of the basic operations of the WDISC is assumed by this document. It does not attempt to explain basic WDISC concepts. See section 1.4 for background reading.

The PSS design described by this document is *as written*, i.e. it is the final version of the design that is actually implemented by the PSS software. Indeed, this document was written some 2 months after the coding was completed. Code revisions through PSS Version 1.2 are incorporated herein.

1.3 Document Organization

This document is divided into two sections and three appendices, as follows:

- Section 1 contains an introduction to this document.
- Section 2 contains both general and detailed PSS design information.
- Appendix A contains a list of the abbreviations and acronyms used throughout this document.
- Appendix B contains instructions for making the PSS a Windows NT service.
- Appendix C contains a brief tutorial of the PSC.

1.4 References

a. WSC Transmission Control Protocol (TCP)/Internet Protocol (IP) Data Interface Service Capability (WDISC) System Requirements, 451-WDISC-SRD 98.

b. WSC Transmission Control Protocol (TCP)/Internet Protocol (IP) Data Interface Service Capability (WDISC) Service Specification, 451-WDISC-SSD 98.

c. WSC Transmission Control Protocol (TCP)/Internet Protocol (IP) Data Interface Service Capability (WDISC) Operations Concept, 451-WDISC-OCD 98.

d. WSC Transmission Control Protocol (TCP)/Internet Protocol (IP) Data Interface Service Capability (WDISC) User's Guide, 451-WDISC-UG 98.

e. Programmable Telemetry Processor for Windows NT (PTP-NT), User's Manual, (No Identifier).

Section 2. PTP Scheduling Server Design

2.1 Design Philosophy

The overriding design philosophy for this project was *Keep It Simple*. It should be noted that this was an external requirement, prompted by a previous implementation of the PSS (named the 'PTP Timer Server'), which suffered from problems resulting in crashes and lockups. These problems were thought to be caused by the large number of threads and/or open sockets it employed. In an effort to minimize potential pitfalls from the PTP Timer Server design, it was therefore decided that the PSS would be kept simple. Accordingly, the number of threads and simultaneous socket connections was kept to a minimum - only 2 threads, and at most 4 sockets (3 in, 1 out).

2.2 Assumptions

The following are operational assumptions that were made which impacted the design of the PSS software to a greater or lesser extent:

a. Requests for simultaneous or closely overlapping service on different PTP I/O boards would be rare.

b. Requests for simultaneous ending and (30 seconds later) starting of service on all three PTP I/O boards would essentially never happen.

c. The starting and ending event times are flexible by a few seconds either way (i.e. early or late) without serious adverse effects.

d. The 'Forward Service' feature would not be used (and although implemented anyway, it was not as strenuously tested as the rest of the code). Forward Service will receive only cursory examination in this document.

e. The only target machines are the Avtec PTP boxes at WSC, running Windows NT Workstation 4.0, Service Pack 4.

2.3 Language

The PSS is written entirely in C++ language. Calls are made, however, to Avtec supplied functions in order to access the I/O boards. These functions were probably written in 'C' (the source code is not available, though).

2.4 Coding Conventions

2.4.1 Files

Each class implementation is typically spread across a header ('.h') and a source ('.cpp') file. The class definition, along with any other definitions that may need to be public (e.g. constants, structures, etc.), are in the header file. Small functions (usually get and put functions) are included in the header file as well (for automatic inlining). Some small classes are wholly contained in their header file (and thus have no source file). The source file contains member function implementations, as well as definitions for any private constants, structures, etc.

2.4.2 Names

All internally defined objects (classes, functions, variables, constants, etc.) should follow the conventions presented in Table 2-1. External objects will, of course, differ from these conventions, but should be obvious nonetheless. Note that multiword lower case and upper case names are separated by underscores; multiword mixed case names are merely run together.

Table 2-1. Naming Conventions

Convention	Example	Use
lower case	level next_time	Local variables Public class members
lower case with initial underscore	_severity_text_ status	Local constants Private class members
mixed case	Logfile CheckRollover	Types Enumerations Classes Functions
mixed case with initial underscore	_Version _EventWindow	Global constants
upper case	MAX_PTP_SERVERS	Preprocessor macros

Single letter variable names are used sparingly; they occur in just two roles:

- the index for a loop, where the index is not used anywhere outside of the loop.
- the argument of a function, where the argument is a generic instance of the type to be operated on by the function.

2.4.3 Comments

There is a large comment block at the top of each file (both header and source). It gives general information about the contents of the file, e.g. a quick description, list of classes, change history, etc. Further comments appear occasionally in the code, as conditions warrant. Only things which are not readily apparent from reading the code are so commented. This design guide should provide any further clarifications necessary.

2.5 Compiler

The compiler that was used to develop the PSS, and produce the final executable installed on the PTP machines, is Microsoft Visual C++ version 5.0 (part of Microsoft Developer Studio 97). The executable was compiled with debug turned off (i.e. a 'release' version). Any questions about the executable that was produced would necessitate checking the documentation for this compiler.

The delivered source has the MS VC++ 'workspace files' included, to make loading and recompiling easier (if MS VC++ is available). These are the PTPSchedServer files with extensions '.dsp', '.dsw', '.ncb', .opt', and '.plg'.

If recompilation by other means should become necessary, follow these guidelines:

- compile all source '.cpp' files to object files.
- link these object files with supplied 'av_ptp.lib' library.
- a windows socket library may be required; the 'ws2_32.lib' library was used for the installed executable. This is standard on Windows NT 4.0, and can be added to Windows 95 with a download from Microsoft.
- the supplied 'av_ptp.dll' file (found in the lib folder, not the source folder), will need to be copied before running the executable. The copy can be placed in C:\Winnt\System32 (preferred), C:\Winnt\system, or C:\Winnt.

2.6 Design Overview

There are no explicit WDISC requirements for the PTP Scheduling Server, other than a few general references to timing and automatic logging. The de facto requirements for the PSS, however, were set by the previous PTP Timer Server implementation. Thus, in order to be a drop-in replacement, the PSS duplicates the functionality and I/O constraints of its predecessor. Other than a very poor set of error messages that can be returned to the PTP Scheduling GUI, there is nothing inherently wrong with these requirements.

The role of the PSS then is to act as a local middleman for the PSG and the PTP I/O boards, since scheduling at the PSG can occur days in advance of need, and the I/O boards only operate in real time. The PSS must accept scheduling requests (events) generated by the PSG, transferred via a socket connection. After verifying the details of the event, a status message (good or bad) must be returned to the PSG. The event information must be kept until the proper time arrives to act upon it. The I/O board specified must then be started with the correct 'desktop'

(control) settings. And at the proper time later, the I/O board must be shut down again. Throughout all of this, messages need to be written to a logfile. Old logfiles should be automatically purged to prevent wasted disk space. The PSS obviously must run continuously, and so should be robust, autonomous, and sparing in resource consumption. Since there are (at least) three customers and three I/O boards, the PSS must be able to handle multiple simultaneous scheduling requests.

Thus, the basic requirements for the PTP Scheduling Server are:

1. Accept incoming schedule requests from PSG and report back results.
2. Keep track of all currently scheduled events.
3. Activate/deactivate scheduled events (i.e. control the I/O boards) at appropriate times.
4. Log all activities to a logfile, with automatic cleanup.

Items 1 and 3 above are processes that need to operate on a continuous, and possibly overlapping, basis. Neither should be made to wait on the other. Item 2 is a data structure requirement. Item 4 is a process that is of lower priority that can be done at leisure.

Given the design philosophy, the preliminary design for the PSS thus emerges. To accommodate items 1 and 3, two separate threads are needed: a 'socket' request thread and a schedule 'timer' thread. The data structure for item 2 must be accessed by both threads, and will need to keep the data sorted by time, so a small 'database' is required. And lastly, logfile routines are needed, again that can be accessed by both threads.

2.7 Design Specifics

The preliminary design given above expanded somewhat during the actual coding of the PSS. The final design encompasses 10 objects (C++ classes), plus the main routine and several global constant definitions. The classes are: Database, Event, Header, Logfile, PTPServer, SchedEvent, SchedSocket, SortedList, Time, and Timer.

The PSS main routine is responsible for setting up the shared class instances (i.e. Logfile and Database), as well as starting the 'timer' thread. The main thread then essentially becomes the 'socket' thread.

When a client makes a connection to the PSS, the request is handled by the socket thread in SchedSocket. A Header record is read to determine the type of request. If the type is a schedule or delete request, an Event record is also read. The Database then processes the request (unless it is a server stop request). SchedSocket then waits for the next request (or exits).

Events are added to the Database as SchedEvent records, which contain extra information used by the timer thread. The records are kept in a SortedList, sorted by their next activation time. The Database tracks the event which is to be activated first (this is not always the first sorted event!). A separate list of events which have been deleted is also maintained. All events which have not yet started are written to a file for disaster recovery (the file is read as the Database initializes).

The timer thread, in Timer, waits for the next activation time of the event to be activated first. Where events closely overlap, event start times can be moved up earlier in time, and stop times can be delayed, to have all I/O boards running on time. When the wait timer expires, a PTPServer starts/stops the proper I/O board. The wait can also be canceled early by the Database, if events have been added or removed (or a server stop has been requested). In this case, the wait time must be recalculated to take into account the changes.

The Logfile records useful operational information, as well as warnings and error messages, from all parts of the PSS. Logfiles are maintained automatically.

The interaction of the classes is shown in Figure 2-1. In the figure, an arrow points toward a class which is used by the class at the other end. Thus, for example, Event is used by both the SchedSocket and Database classes, and in turn uses the Time class. The double outlined classes correspond to the two PSS threads.

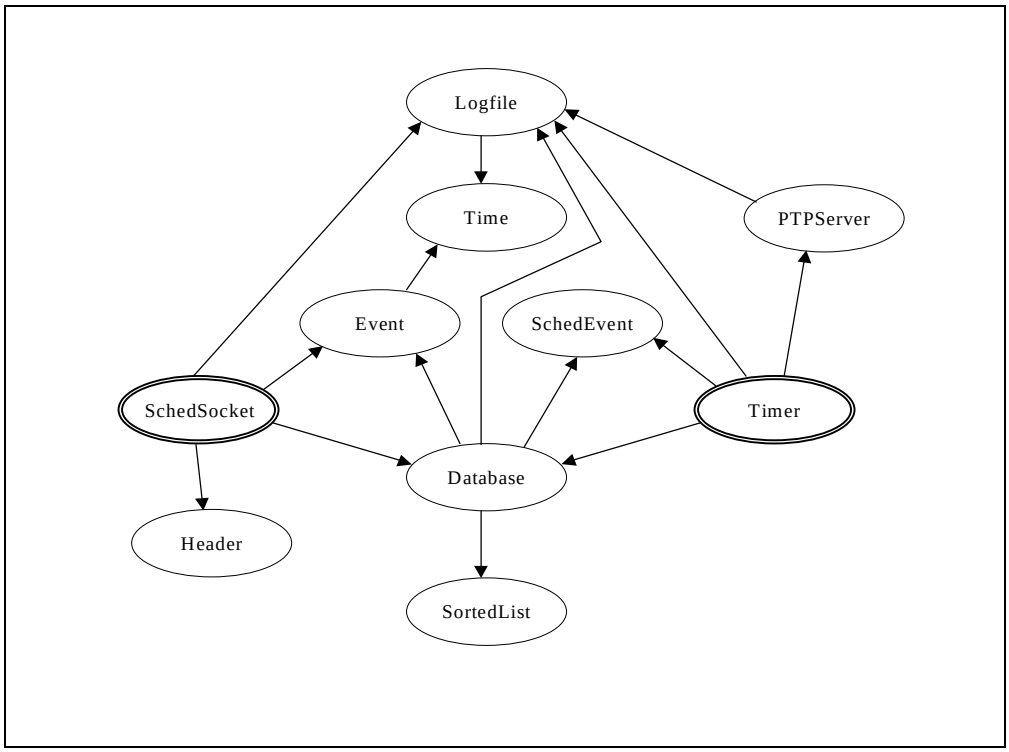

Figure 2-1. PSS Class Interactions

2.8 Implementation Details

The following sections give a detailed description of the inner workings of the PSS.

2.8.1 Definitions

Files: definitions.h

This is a collection of globally useful constants and template functions. Also, any constants that may need to be changed in the future (e.g. timing values and directories) are collected there, for ease of finding them. The values currently defined are given in Table 2-2.

2.8.2 Main Routine

Files: main.cpp

The PSS main routine is responsible for setting up all shared class instances, as well as starting the timer thread. The main thread then essentially becomes the socket thread. Figure 2-2 shows a flow diagram for the main routine.

The first thing that is done is a test to see if the PSS is already running. If so, execution is halted, since there can only be one listener on the port for PSG socket connections. This test is accomplished by creating a named system event object. Only one such object may exist. Any errors return a code, but do not write anything anywhere.

Next, the Logfile instance is created and set up. If there is a problem, again an error code only is returned. Otherwise, error messages can now be written to the logfile.

The Database instance is then created. Any errors are logged and the server exits.

Next, the timer thread is started. Pointers to the Logfile and Database instances are passed to the new thread, so both threads can have access to the same instances.

Finally, a SchedSocket instance is created and given control of the main thread. Since the server is supposed to run indefinitely, control does not normally return. If control is returned, then a server shutdown has been requested via the PSC. In this case, everything is destroyed and the server exits.

The 'old' thread creation routine, _beginthread(), was used due to the following caveat found in the MSVC++ documentation:

A thread that uses functions from the C run-time libraries should use the _beginthread and _endthread C run-time functions for thread management rather than CreateThread and ExitThread. Failure to do so results in small memory leaks when ExitThread is called.

The few references to C library routines (mostly sprintf calls) in the PSS code could have been eliminated. However, the Avtec supplied library is probably C code, and may contain C library references. Although the PSS does not exit any threads (unless the entire server is exiting), it is probably better to be safe than sorry.

Table 2-2. Definitions

Type	Name	Value	Units	Description
type	String	const char *		Shorthand for a constant char pointer
constant	_Version	"1.2"		Program version number
constant	_DesktopDirectory	"C:\Ptp_User\Desktops"		Directory where desktop definition files are stored
constant	_LogfileDirectory	"C:\Ptp_User\SchedServer\Logs"		Directory where program logfiles are stored
constant	_DatabaseDirectory	"C:\Ptp_User\SchedServer\Data"		Directory where program database files are stored
constant	_MinuteSeconds	60	seconds	Number of seconds in one minute
constant	_HourSeconds	60 * _MinuteSeconds	seconds	Number of seconds in one hour
constant	_DaySeconds	24 * _HourSeconds	seconds	Number of seconds in one day
constant	_ListenPort	3999	seconds	TCP/IP port that server listens on
constant	_EventTolerance	15	seconds	Minimum time between events on same I/O board with same desktop
constant	_EventWindow	72 * _HourSeconds	seconds	Maximum time in future an event can be scheduled
constant	_LogfileRetention	14 * _DaySeconds	seconds	Age of logfile before automatic deletion
constant	_SocketTimeout	60	seconds	Maximum time spent reading data on socket
template	Min(a,b)	a < b ? a : b		Minimum of two values
template	Max(a,b)	a > b ? a : b		Maximum of two values

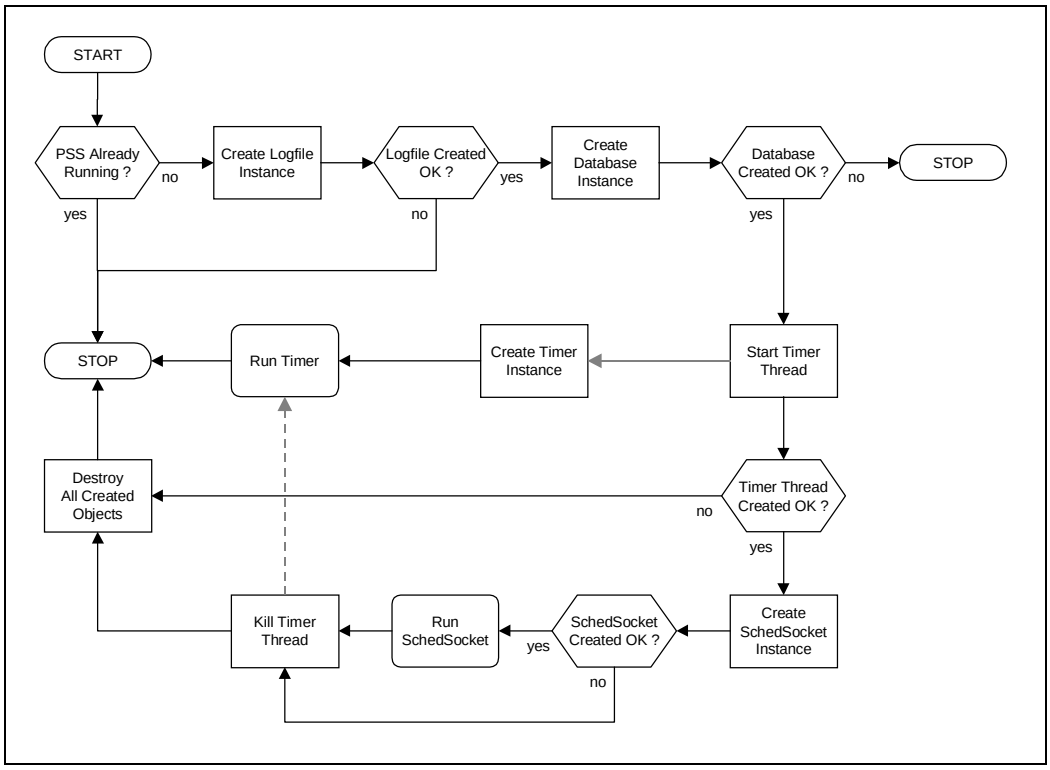

Figure 2-2. Main Routine Flow Diagram

2.8.3 Database Class

Files: database.h, database.cpp
Instances: 1

The database is responsible for handling all communication between the socket and timer threads. Most of this is legitimate database work, but some is extra that doesn't really belong here (but had no better place to go).

The database defines the class SchedEventList, which is really a typedef for the SortedList template with the SchedEvent class. Thus we have a sorted list of SchedEvent pointers; this will form the core of the database. The overloaded validation function (IsValid) checks for overlapping event times on the same I/O board. The comparison function compares two events based on their next activation times.

Two SchedEventLists are maintained by the database. The main list (_schedule) holds all scheduled and active events. The second list (_interrupt) holds only those active events which have been requested to be deleted, if any. Deletion cannot take place immediately, because the timer thread has to be informed so that the I/O boards can be shut down properly.

A critical section variable (_crit) is used to prevent concurrent access by the two threads. The critical section can be requested and released externally, to allow for complex access to the database.

The database provides a signal handle, which is used by the timer thread when waiting on events. In this way, the database is able to interrupt the timer thread during a wait if there are new or deleted events to process.

The first active event is actually determined by the Timer, however it is more properly kept by the database for future reference. For more information on the first active event, see section 2.8.12 on the Timer class below.

The stop flag (_stop) really has nothing to do with the database; it is located here for convenience only. Otherwise, another inter-thread communication mechanism would have to have been set up. The flag is set when a shutdown of the server has been requested by the PSC.

All non-active events are written to a file on disk in case of abnormal server shutdown. The database is preloaded at startup with any events in the file which are still valid (i.e. not past their start time). As an extra measure of protection, any existing database file is kept as a backup while writing the new one; in this way if a failure occurs during the write, something useful may still be regained. The backup is deleted to indicate a successful write. The read routine takes this into account, and thus looks first for the backup file, and then the normal database file.

2.8.4 Event Class

Files: event.h, event.cpp
Instances: many

The Event class is actually the data structure sent from the PSG to the PSS with the scheduling information for an event. Member functions then give access to the internal data fields and provide rudimentary validity checking.

2.8.5 Header Class

Files: header.h
Instances: many

The Header class is another data structure; this one is used for sending command information from the PSG to PSS, and status information from the PSS back to PSG. Again, member functions give access to the internal data fields, and provide simple error checking.

2.8.6 Logfile Class

Files: logfile.h, logfile.cpp
Instances: 1

The Logfile class is responsible for maintaining a set of logfiles on disk. Either thread may submit logfile entries without problem; the writing is protected (via critical section) to prevent interleaved entries. Logfile entries are automatically timestamped and typed (i.e. information,

warning, or error). The logfiles are maintained automatically, with rollover to a new logfile each day, and purging of old logfiles after a set period of time (currently, 14 days).

On startup, the logfile directory is checked for write permission by opening a test file (testlog.txt). The test file is not deleted. It is therefore perfectly normal for this empty file to exist in the logfile directory.

Submissions can be made to the logfile in several formats: as a plain string, as a string with an embedded integer (using sprintf %d notation), or as a string with an embedded string (using %s).

Logfiles are only checked for automatic rollover and/or deletion when something needs to be written. This means that during periods of no server activity (or at least none worth logging), the open logfile will not rollover, and logfiles older than 14 days will stick around. Everything will be updated at the next logfile write.

2.8.7 PTPServer Class

Files: ptp_server.h, ptp_server.cpp
Instances: 1

Control of the I/O boards is actually accomplished through a socket connection to a separate Avtec server; however, an Avtec supplied library encapsulates and abstracts away from the underlying communication.

The PTPServer class serves as an easy interface to the PTP I/O boards, since many Avtec library commands must be issued for each PSS 'highlevel' operation (e.g. board startup, start forward service, stop forward service, and board shutdown). Logging is also provided.

2.8.8 SchedEvent Class

Files: sched_event.h, sched_event.cpp
Instances: many
Base Class: Event

The SchedEvent class is the data structure used by the database to store scheduled event information. It is derived from the Event class, adding members to track the stage of the event and its PTPServer connection.

The event stage indicates which event time is next to be activated: event start, forward service start, forward service stop, or event stop. Those events in the *event start* stage are therefore merely scheduled; all other stages indicate an active event.

2.8.9 SchedSocket Class

Files: sched_socket.h, sched_socket.cpp
Instances: 1

The SchedSocket class handles the duties of the socket request thread. It sets up a port listener, and waits for incoming socket connections, which indicate a request from the PSG/PSC. When one arrives, a client socket is opened, and communication established. A single command from the client is processed, and the client socket is then closed. The connection times out after approximately 60 seconds, to prevent server lockups in the case of network problems. Figure 2-3 shows a flow diagram for the SchedSocket.

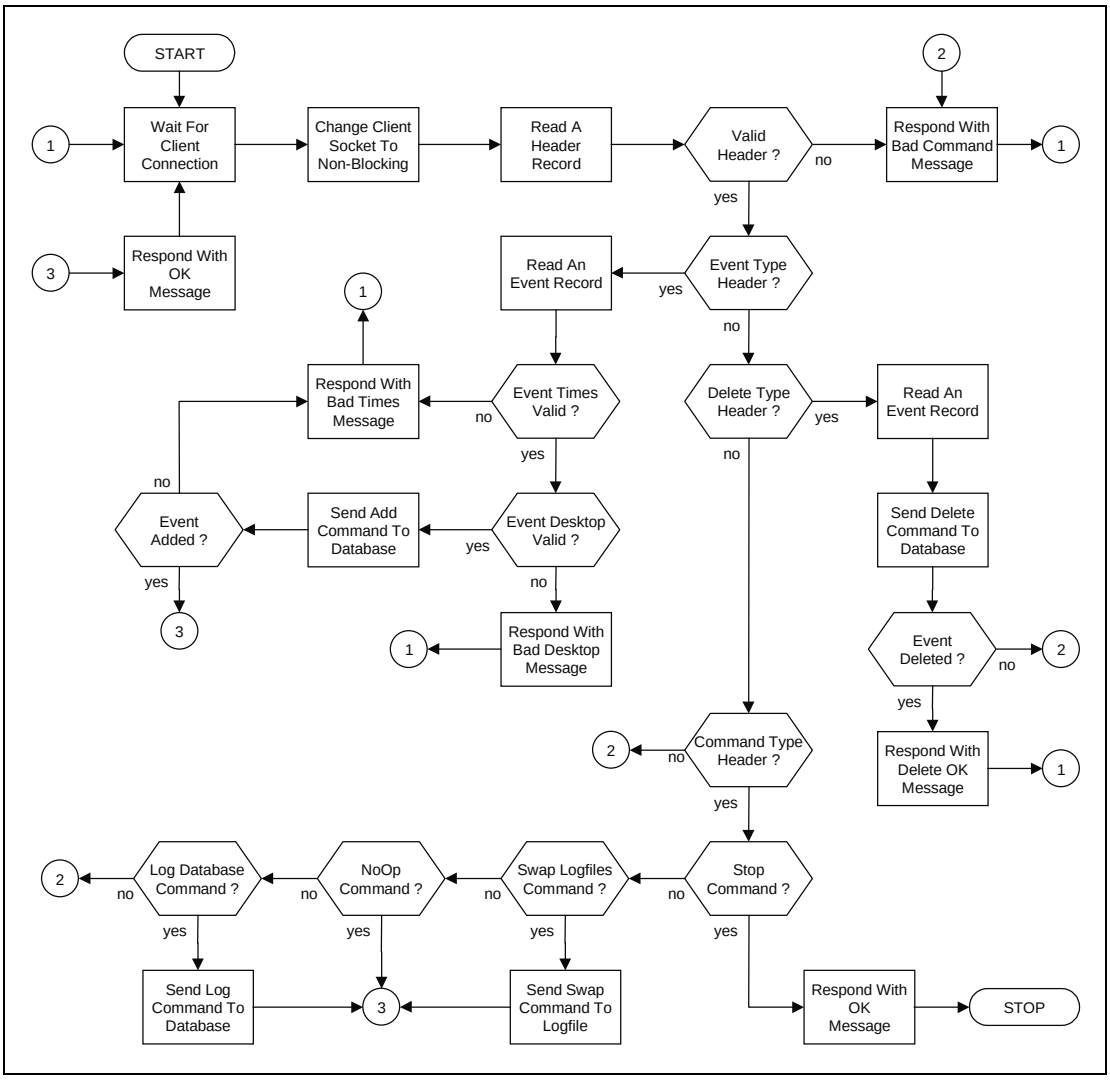

Figure 2-3. Scheduling Socket Flow Diagram

Initially, data is read from the socket directly into a header record. The header can then be queried for its type. Commands are only generated by the PSC; there is no way of sending them from the PSG. The valid commands are: no operation, stop server, log database, and swap (i.e. rollover) logfiles. The first command is easily handled internally, the next two are sent to the Database, and the last is sent to the Logfile.

Data for scheduling requests is read into an event record, validated, and then processed by the database (with extra validation for overlapping events).

Delete requests also require reading data into an event record. The information is then passed to the database for processing.

The server responds to the client with a limited vocabulary: OK, invalid command, desktop file not found, invalid time, forward service file not found, and deletion OK. This means that many errors are unfortunately lumped together under the *invalid command* and *invalid time* responses.

2.8.10 SortedList Class

Files: sorted_list.h
Instances: 1

The SortedList class is really a template; it is the basis for SchedEventList, which is used by the Database class. SortedList provides the functionality for a list of pointers which is kept in a special, "sorted" order. The method of sorting is up to the class built from the template. Items can be added to, removed from, or copied from the list in several ways (e.g. first, last, position, previous, next, etc.).

2.8.11 Time Class

Files: xtra_time.h, xtra_time.cpp
Instances: many

The Time class is a simple, multiformat time representation. A time value can be set or retrieved in the standard 'time_t' and 'struct tm' formats, as well as retrieved in a character string format.

2.8.12 Timer Class

Files: timer.h, timer.cpp
Instances: 1

The Timer class is responsible for controlling the PTP I/O boards at the scheduled times. This is simple in concept, until an ever-changing list of events, overlapping event times, and altered event timing are added to the picture. Figure 2-4 shows a flow diagram for the Timer.

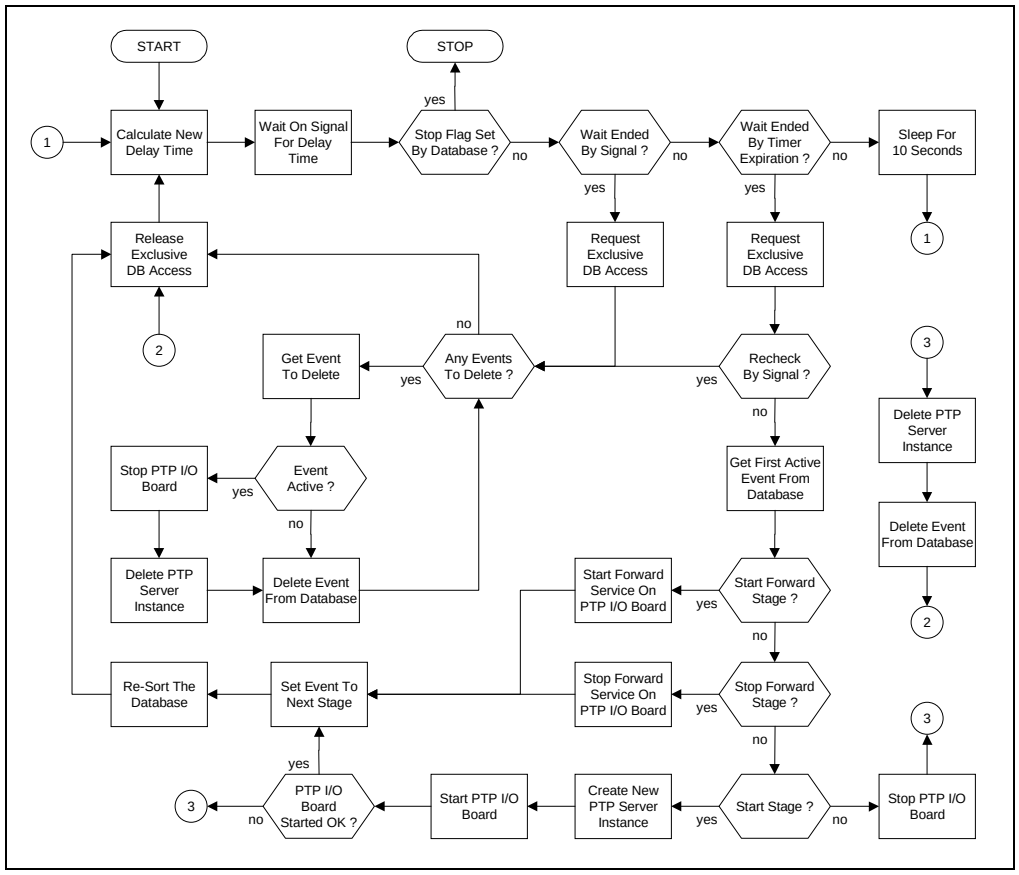

Figure 2-4. Timer Flow Diagram

Generally, the timer merely needs to wait until the next activation time of the first active event. A simple sleep call would suffice for this. But the timer must also be notified when new events are scheduled, or existing ones deleted, as this may change the first active event (and thus the wait time). So a combination timer-signal device is required. Fortunately, signal handles can be waited on until signaled or until a timer expires. The signal is owned by the database, since it will be doing the signaling.

After wakeup (via either method), the timer checks to see if a shutdown request has been made, and if so, exits. This is a fast shutdown, and does not change the state of the PTP I/O boards in any way. It does, however, guarantee that the timer is not interrupted in the middle of controlling an I/O board, which could cause problems (with the board, or the controlling DLL).

The wakeup method is then checked. In either case, the timer requests exclusive access to the database. If the timer expired, the signal is quickly rechecked to make sure no new events have just arrived. If there are none, the first active event is then processed according to the stage it is currently in.

If the wakeup is via the signal, the database is checked for deleted active events. If any are found, the corresponding I/O board is shut down. Then, exclusive access to the database is released, and the delay is recalculated for the next wait.

In order to understand the mechanics of the scheduling delay calculation, it is first necessary to understand the details of event timing. First, it should be noted that although an event specifies activation at a precise point in time, the processing of that activation is not instantaneous (start requests may take up to 5 seconds, stop requests even longer). Furthermore, waking from a timer expiration is not necessarily exactly on time, depending on other OS activities. Therefore, associated with the true event specified activation time (event time) are two other offset time values: a buffer to hedge for imprecise timer expiration (buffer time) and the actual processing time (execution time). These offsets are constants that are set to reasonable values, based on testing. This gives rise to two extra time points: the buffered time and the active time. Buffered time is the event time, adjusted by the buffer time; this is backwards in time for start requests and forwards for stop requests (i.e. the event should start a little early and run a little late). Active time is the actual time for action, taking into account the execution time (for start requests). All of this event timing is depicted in the upper half of Figure 2-5.

Generally speaking, a stop request can be delayed a short while without any problem. Start requests, however, cannot be delayed. Therefore, if multiple event start times overlap, something must be done so that all can finish by the actual event time. This leads to the concept of a multi-start, which is depicted in the lower half of Figure 2-5. Here, start event are 'stacked' in time by their execution times, with a single final buffer time. This concept does not apply to stop events.

Figure 2-6 is a flow diagram which shows how the scheduling delay is calculated. To understand this, the first active event needs some explanation. Although the database maintains the list of events in order based on their next activation time, this is not necessarily the order of actual activation, due to altered event timing, as detailed above. Thus, the event which is truly next to need activation (the first active event) needs to be determined. This is the true job of delay calculation; once the first active event is known, the delay is easily determined.

In order to find the first active event, the database is scanned in the reverse of its sorted order.
At first this may seem odd, since logically the first active event should be one of the first few events in the database. While this is true, it is much easier to build multi-start event timing in order going backwards in time from the shared event time. Also, it would be more difficult to determine where stop requests are being pushed beyond a corresponding start request for the same I/O board. Since the database is never expected to get very large, this extra overhead (reading the entire database each time) should be minimal.

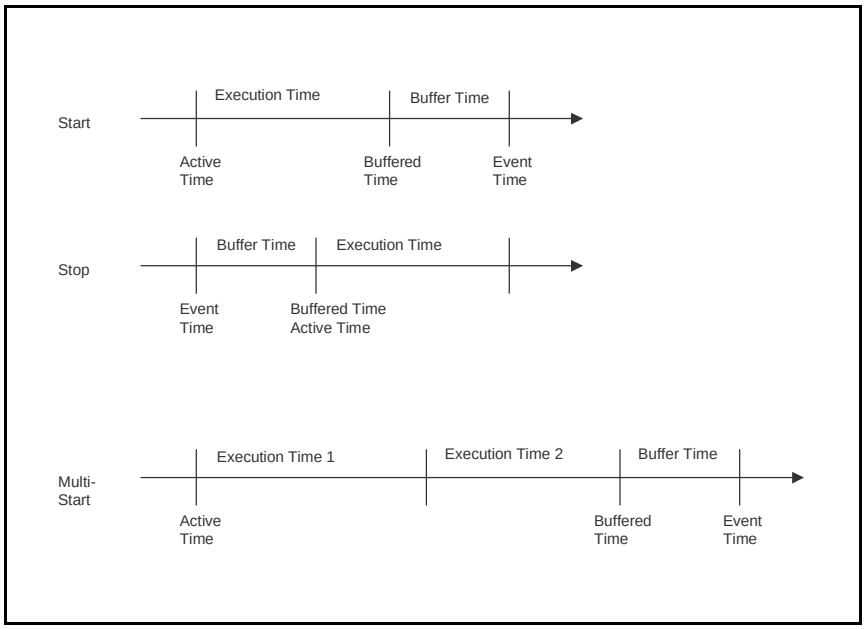

Figure 2-5. Detailed Event Timing

Each event from the database is compared to the current first active event. Events which can complete execution before the current first active time simply become the new first active event. Otherwise, the event request type must be checked. For start requests, a multi-start list is built, since the events overlap (they have to, due to the sorting of the database and the failure of the first test). For stop requests, the multi-start list is checked; if it is empty or a corresponding start event is found, the stop request is the new first active event. Otherwise, the stop request is ignored (for now). When scanning of the database is complete, the active time of the first active event is compared to the current system clock time to determine the scheduling delay.

The frequent recalculation (from scratch) of the scheduling delay should limit the effects of any slippage in time due to execution overruns.

2.9 Known Bugs

There is one known bug in the PTP Scheduling Server (version 1.2).

The PSS returns an incorrect code to the PSG after successfully processing a 'delete event' request. The code returned is 'OK', but should be 'deletion OK'. This makes no operational difference to the PSG, however a possibly misleading entry is written to the PSG logfile.

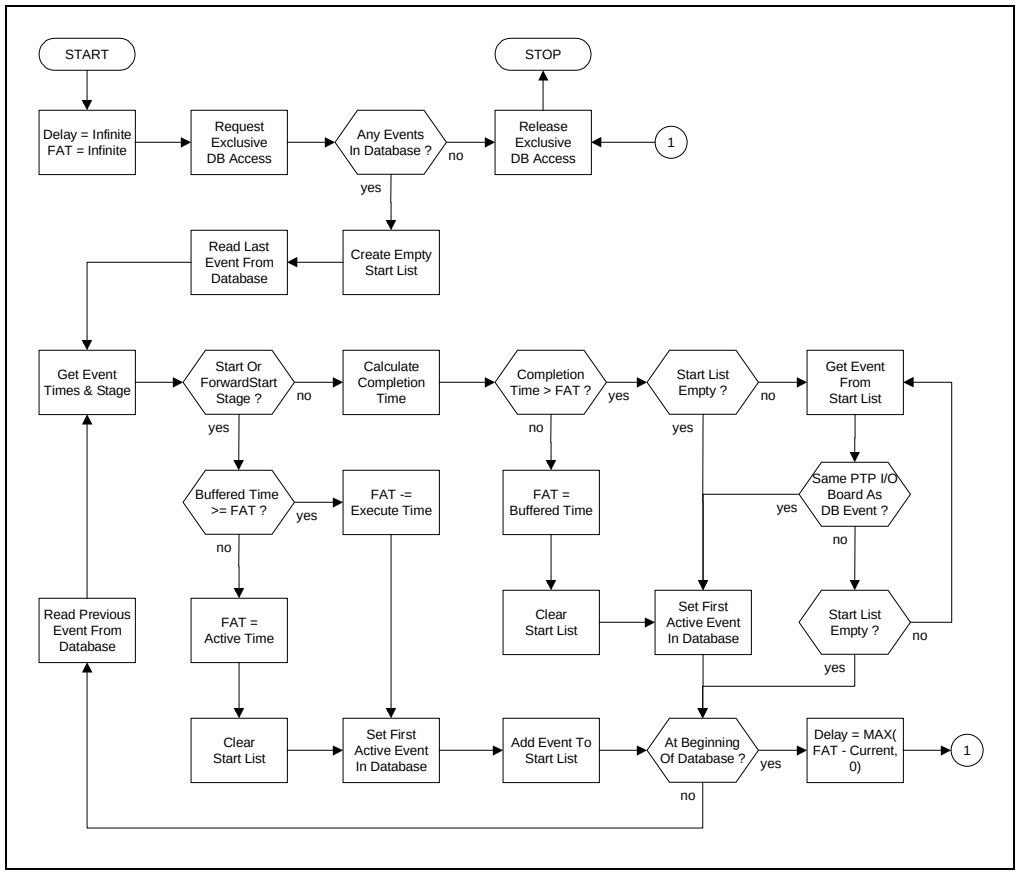

Figure 2-6. Scheduling (Delay Calculation) Flow Diagram

2.10 Missing Features

Due to time constraints during development, two features (not required) of the PSS were never implemented:

1. Reading a configuration file for easy changes of the constant values found in the definitions.h file. This would allow parameter changes without incurring a code recompile.
2. A ‘graceful’ shutdown request that would wait until nothing is happening on any of the I/O boards before shutting down the server. Without a change to the PSG <-> PSS interaction, this would only be possible from the PSC, as the current 'quick' shutdown request is.

Appendix A. Abbreviations and Acronyms

Avtec	Avtec Systems Incorporated
EO-1	Earth Orbiter-1
FUSE	Far Ultraviolet Spectroscopy Explorer
GP-B	Gravity Probe B
GUI	Graphical User Interface
IONET	IP Operational Network
IP	Internet Protocol
MOC	Mission Operations Center
NASA	National Aeronautics and Space Administration
NCC	Network Control Center
NISN	NASA Integrated Services Network
NMP	New Millennium Program
OS	Operating System
PSC	PTP Scheduling Client
PSG	PTP Scheduling GUI
PSS	PTP Scheduling Server
PTP	Programmable Telemetry Processor
TCP	Transmission Control Protocol
WDISC	WSC TCP/IP Data Interface Service Capability
WSC	White Sands Complex

Appendix B. Windows NT Service Setup

The PSS executable is intended to be used on the PTP machines as a Windows NT service. Since it was written as a 32-bit console application, however, it is necessary to use the ServAny utility as an intermediary. ServAny is a proper Windows NT service executable which is capable of starting another console or windows application as a child process. The documentation that comes with the ServAny utility is a bit cryptic, so the following are step-by-step instructions for setting up the PSS executable as a service:

1. Locate the ServAny.exe and SrvInstW.exe utilities on the system. They do not come standard with Windows NT, but are available from Microsoft as part of the Windows NT Resource Kit.
2. Open an MS DOS Command Prompt.
3. Use SrvInstW to install ServAny as a service. Run srvinstw.exe. On each page of the wizard, do the following:
   - Click 'Next' to install a new service.
   - Click 'Next' to install on the local machine.
   - Enter 'PTPSchedServer' in the box and click 'Next'.
   - Enter C:\full\path\to\servany.exe in the box and click 'Next'.
   - Click 'Next' to use the service as its own process.
   - Click 'Next' to use the System Account without desktop interaction.
   - Click 'Next' to use automatic service startup.
   - Click 'Finish' to complete the installation.
4. Run regedit.exe to edit the registry entry for the new PTPSchedServer service just created:
   - Find the following key: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PTPSchedServer
   - Edit the DisplayName string value to read 'PTP Scheduling Server'.
   - Under the PTPSchedServer key, create a new key named 'Parameters'.
   - Under the Parameters key, create a new string value named 'Application'.
   - Edit the Application string value and enter the full pathname to the PSS executable (with extension).
   - Exit the registry editor (changes are saved automatically).

The new PSS service will be started automatically the next time Windows NT is restarted. In the mean time, it can be started manually from the 'Services' control panel. Highlight the PTP Scheduling Server entry in the list, and click the 'Start' button.

Appendix C. PTP Scheduling Client Operation

The PTP Scheduling Client (PSC) was developed as a testing tool for the PSS. The 'official' client used by the MOCs and the NCC is the PTP Scheduling GUI (PSG); the PSC duplicates the functionality of the PSG, and includes additional features as well. The PSC is, however, a command line driven program with minimal error checking, and thus is not suitable for general use. It might be useful for future testing or in emergency situations, though, so its operation is briefly described here. It is installed on the PTP boxes at WSC.

The PSC must be run from an MS DOS command prompt. Upon startup, it requests the IP name or address of the machine where the PSS is running. The default is the local machine. Next a menu of commands is displayed. Commands are entered as a single character, followed by pressing 'Enter'. The displayed number/punctuation can be used, as well as the first letter of the command (e.g. List local PTP event(s) can be entered as '7' or 'L' or 'l').

1: Test connection to server

This command merely sends a NoOp command to the PSS, which should reply with an OK message. The result is reported on the screen. This is used to test the connection to the server.

2: Copy server event(s) to logfile

This command instructs the PSS to dump its internal database of scheduled and active events to the logfile. The success of the request is reported on the screen.

3: Force server logfile rollover

This command instructs the PSS to close the current logfile, and open the correct logfile for the current day. This will probably be the same logfile it just closed if there has been any server activity recently. Data is always appended to the logfile. The success of the request is reported on the screen.

4: Enter a local PTP event

This command allows the user to enter the data for a PTP event. First an event slot (there are nine total) must be selected. The PSC can keep track of up to nine events. Then the current time is displayed. Next, the user is asked to enter the server port (for the I/O board), the desktop filename (full pathname is not needed), the event start time, and the event stop time. Also, the user is asked if there is forward service; if so, the forward service start and stop times are requested. Remember, almost no error checking is done on user input. This was actually intentional, in order to test the PSS. At any of the prompts a '!' can be entered to abort the data entry. Nothing is sent to the PSS.

5: Schedule an event with server

This command is used to send a local PTP event to the PSS for scheduling. The event slot must first be selected. The results of the request are reported on the screen. The error checking of the server will reject many values accepted by the PSC. Unfortunately, the list of valid messages from the server is short, and uninformative.

6: Delete a scheduled event

This command is used to delete a previously scheduled PTP event from the PSS database. First, the event information must be entered locally (via '4') in an event slot. This command then asks for the event slot. The success of the request is reported on the screen.

7: List local PTP event(s)

This command simply lists on the screen the data that has been entered (if any) for each of the local PTP event slots.

8: Wipe clear local PTP event(s)

This command deletes all data that has been entered for each of the local PTP event slots. Nothing is reported on the screen.

9: Kill the server quickly

This command instructs the PSS to shut itself down as quickly as possible. If the PSS is idle, this will be instantly. If the PSS is busy processing an event start or stop, the action is completed before shutting down. This is to avoid problems with the I/O boards. Note that an event can be interrupted between the start and stop; in this case the I/O board is left running. The success of the request is reported on the screen.

+: Add 1 min to local PTP event(s)

This command simply adds 1 minute to all of the times that have been entered for each of the local PTP event slots. This is useful mainly while testing, to send a suite of events to the server over and over again. Nothing is reported on the screen.

?: Help (redisplay this menu)

This command simply redisplays the list of available commands on the screen.

!: Quit

This command quits the PSC program. Nothing is sent to the PSS.